On The Expression Of Interaction Properties Within An Interactor Model

Panos Markopoulos

Dpt. of Computer Science, QMW College, University of London
Mile End Road, London E1 4NS

Abstract. This paper introduces a formal model for the description of interactive systems based on the interactor model of [15, 17]. Similarly to that model, it is intended to be used constructively for building specifications of interfaces as compositions of interactors. Changes are brought about to two aspects of the model: firstly, a modularised representation of control information is achieved which supports the independent description of the data transforming behaviour of the interactor and of the temporal constraints imposed on that behaviour. Secondly, distinct representations of 'result' and 'display' data handled by an interactor are related within a process algebraic framework, allowing the expression of usability related properties of interaction.

1 Introduction

This paper presents an approach to modelling interactive systems based on the concept of the interactor. The model introduced is a variation of the interactor model of Paterno' and Faconti [15, 17] which is referred to below as the Pisa interactor. The changes introduced allow for the separation of the interactor specification into two components: one implements the data operations performed by the interactor and the other embodies the dialogue control elements of its specification. This separation is preserved when forming compositions of interactors. The modeller may choose between two different views of the interface specification: an architectural view where the interface is specified as a composition graph of simple interactors and a 'design oriented' view where a 'centralised' dialogue control component may be inspected and modified. Another consideration in forming the model has been to reproduce, at this more 'concrete' level, descriptions of the properties of interactive systems introduced in [1, 9, 19]. To this end, it is essential to distinguish between those aspects of the state of the interactor which are relevant to the application and those which are shown to the user.
The meaning and some of the different uses of the interactor concept are introduced in section 2. An informal description of the proposed model, in section 3, brings out the motivations and concerns which led to variations from the Pisa model. A detailed account and formal description of the interactor model is given in sections 4 and 5. Comparisons are also drawn with the Abstract Interaction Objects model by Duke and Harrison [6], which will be referred to below as the York interaction model.
The LOTOS formal specification language [2] is used to describe interactors. LOTOS has a process algebra component and a data typing component, a duality which renders it a powerful notation for the specification of interactive graphics systems. A brief

example is presented in section 6. In section 7, properties of interactive systems are described in terms of the observable behaviour of interactors. In section 8, the contribution of this research is summarised and its relationship to current and future research activities is outlined.

2 Interactors

Interactors are primitive abstractions used in the description of interactive systems. They can be thought of as software architecture abstractions similar to objects in object oriented programming. Definitions vary with their intended use. Faconti defines an interactor as an entity of an interactive system capable of reacting to external stimuli; it is capable of both input and output by translating data from a higher level of abstraction to a lower level of abstraction and vice versa [7]. Input and output functionalities and events are distinct and the display is simply modelled by the display events offered by the interactor. The Pisa model proposes the use of interactors as a design construct by modelling the interface software as a graph of communicating interactors [17]. At the lowest level they interact with the user and at the highest levels they communicate with the application.

Duke and Harrison view the interactor as a component in the description of an interactive system that encapsulates a state, the events that manipulate the state and the means by which the state is made perceivable to the user of the system [6]. The York interactor extends the software engineering notion of the object in that it maintains distinct representations of the object's internal state and its display. The York interactor does not distinguish between layers of abstraction of data or the source of events. Its intended use is primarily analytical.

Structuring interface software in terms of interactors is similar to conceptual architecture models for interactive software like PAC [3] and implementation architectures like MVC [10]. Interactors have been used also as an implementation construct for the input model of the GARNET [13] user interface development environment, encapsulating device and application independent behaviours of interactive components.

3 Informal introduction of the ADC interactor

The Abstraction Display Controller (ADC) model distinguishes two aspects of the interactor description: the data behaviour i.e. the data operation it supports and the control information, pertaining to the temporal ordering of its behaviour. The former is modelled by the Abstraction and Display Unit (ADU) whose function is that of translating data in two directions between levels of abstraction and the latter is modelled by the Controller component (C). The ADU maintains a state representation which will be called the *abstraction* and a representation of its output status the *display*. Conceptually, the abstraction is similar to the model of the MVC architecture [10] and the abstraction of the PAC model [3]. The display is similar to the MVC notion of a view.

Data from the user or the application side may be input to the ADU which will update its status information accordingly. The ADU may output its display status towards the user side or an interpretation of the abstraction status, the *result*, to the application side. Communication of data is modelled as LOTOS events offered over ‘gates’ of the ADU, each of which is dedicated to a particular direction of communication.

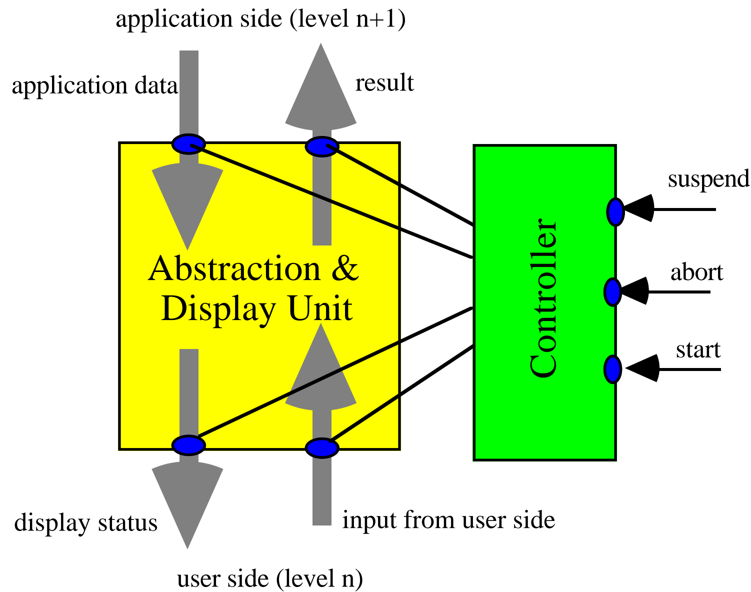

Fig. 1. A schematic view of the ADC interactor

The ADU does not impose constraints on the temporal behaviour of the interactor: it always offers events on all its gates. At any instance during the course of interaction, the ADU offers to 'buffer' information that flows between user and application, converting the data to the appropriate representation in each direction. What may or may not be an allowable sequence of events is defined by the *controller* component in terms of constraints on the behaviour of the ADU. The composition of the ADU and the controller forms the ADC interactor, shown in figure 1. Thick arrows represent the input and output of data to the ADU. Control events, represented with single lines, will enable or disable the communication of data over the gates of the ADU. The controller also describes suspension and termination of the interactor operation.

This modular description of control information aims to make the ADC interactor more usable as a design oriented representation. Temporal ordering constraints on the behaviour of the interactor are represented independently of the data handling behaviour of the interactor, as a set of constraints applied to its externally observable behaviour. This is in contrast to the Pisa interactor where the specification of the temporal ordering of events is by the composition of lower level entities. The difference is that between a resource oriented specification and a constraint oriented specification [20]. One advantage is that it allows for the easy inspection and customisation of the interactor by applying constraints on its observable behaviour.

Interactors are derived as instantiations of the ADC model by defining the operations on the data in the fashion introduced in [15]. Interfaces are modelled as compositions of interactors, using the composition operators of LOTOS. Definition of the start, suspend and abort behaviours of the interactors allows interfaces to be configured dynamically.

The composition of two (or more) interactors has two facets: the composition of their effect on the data transmitted, which is achieved in practice by 'piping' the data from one to the other, and the composition of their control specifications. The advantage of

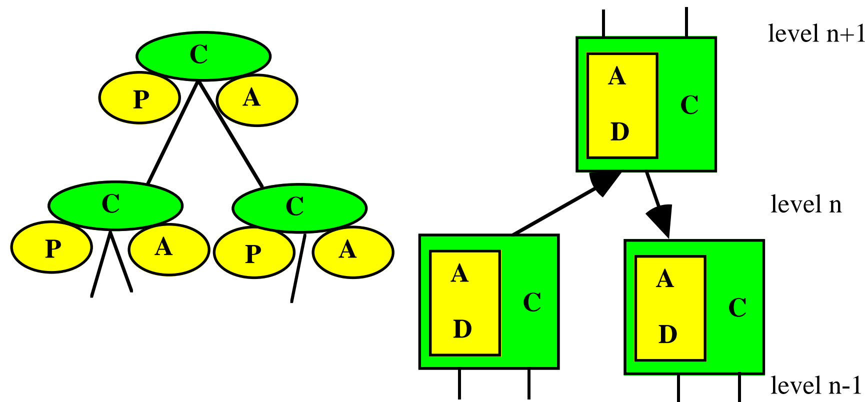

Fig. 2. Although similar to PAC, the controller has a different role in the ADC model.

the modularity introduced with the ADC model should become more evident when compositions of interactors are associated with a single separable control component. In other words, the effect of the composition operators on the controller component can be abstracted in a new controller component.
The components of the interactor identified above (abstraction, display and controller) are reminiscent in purpose and naming to those identified by the conceptual software architecture PAC [3]. PAC is mentioned because it supports a distinct control component (although the model did not prescribe how the control information was to be represented in this component). There is an important difference between the ADC and PAC interactors: the PAC controller handles all communication between the presentation and abstraction components and translates data between the two formalisms. Such communication is 'hidden' from the ADC controller which simply imposes external 'dialogue' constraints on their operation (figure 2).

4 The Abstraction and Display Unit (ADU)

The ADU mediates between a higher level of abstraction and a lower level of abstraction. It can receive information from either side, process the information and pass it on to the other side. It maintains the local state of the interactor which consists of the *abstraction* and the *display*. The ADU offers the following operations on these state components:

- *input*: it constructs a new abstraction status by interpreting data input from the user side with respect to the display status and the abstraction status.
- *echo*: it constructs a new display status by interpreting data input from the user side with respect to the display status and the abstraction status.
- *result*: an interpretation of the abstraction status is constructed; this may be sent to the application side.
- *render*: the display status is updated with respect to data received from the application side. Thus the application (or higher level interactors) may impose constraints on the graphical appearance of the interactor in question.
- *receive*: the abstraction status is updated with respect to data received from the application side.

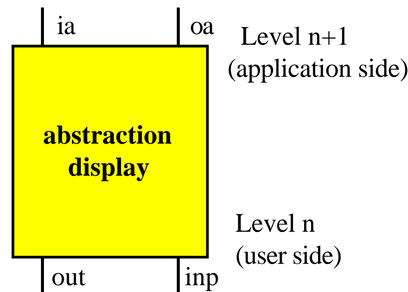

Fig. 3. The ADU process, its gates and state parameters

The input function defines the syntactic as well as semantic dependence of user input to the current output. Also we note that an interactor may receive from higher levels of abstraction, both data that will modify the abstraction information it holds and data that will modify the description of its display.

The overall concept is similar to the Pisa model with two important exceptions:

(a) Trigger events, which signal when the data held by the interactor should be input towards the application or output towards the user, are not distinguished from other input events from the user or the application side. For the ADU a trigger event is just another input event; the trigger behaviour is seen as a control structure so it is described in the controller component (which is discussed in the next section).

(b) In the Pisa model, state information which in the ADC model is held by the *abstraction* parameter is divided between the collection and measure processes. The Pisa interactor communicates user input to higher levels of abstraction and display information in the opposite direction, but does not capture the same notion of local state as is desired in this study.

Contrasted to the York interactor, we note the constructive, rather than analytic, nature of the ADC model and the different perspective offered by a process algebraic framework for the definition of the model. The York model represents explicitly the relationship between the state internal to the interactor and the part of the state that is displayed to the user. The York interactor does not differentiate trigger events either; in fact it does not distinguish between events for input, output etc.; this kind of control information is captured indirectly by the effect of events on the local state of the interactor.

Having described the functionality of the ADU, its operational description can be given. In the process algebraic framework of LOTOS, the ADU is considered as a recursive process that continuously offers events over four gates. The ADU may:

- receive data from the application side via gate *ia*. The data is interpreted using functions *receive* and *render*.
- output to the lower level its display status via gate *out*.
- receive data from the user side via gate *inp*. This data may modify the currently held display and abstraction status, using functions *echo* and *input* respectively.
- send an interpretation of the currently held state information to higher levels of abstraction via gate *oa*, using function *result*.

Notably there is an asymmetry in handling the abstraction and the display data. The display is transmitted as it is, while an interpretation of the abstraction is transmitted. Regarding the display status which is observable via the gate *out*, it is important to distinguish between what is actually displayed and information that has not been

displayed yet. For example, consider a sequence of consecutive input events that have not been echoed (given that there is no built in constraint for immediate echo of the input). These input events are interpreted with the last display status offered on the gate *out*. Thus, two variables are needed to describe the display: one holds the last data offered on the output gate (held below by formal parameter *ds*) and the other, the data that will be displayed with the next output event (formal parameter *dc*). This is a necessary complication since there is no constraint that output will occur immediately after an input or an output trigger.
A formal specification of this model in LOTOS is presented below. The data type *ad* defines the generic to interactors operations on the data, as described above.

```
type ad is
sorts
      inp_data, abs, disp, ia_data, oa_data
opns
      input    :    inp_data, disp, abs     ->    abs
      echo     :    inp_data, disp, abs     ->    disp
      render   :    disp, ia_data           ->    disp
      receive  :    abs, ia_data            ->    abs
      result   :    abs                     ->    oa_data
endtype
```

The behaviour of the ADU can be described in LOTOS as follows:

```
process adu[inp, out, ia, oa](a: abs, dc, ds: disp) : noexit :=
      oa!result(a);        adu[inp, out, ia, oa](a, dc, ds) []
      out!dc;              adu [inp, out, ia, oa](a, dc, dc) []
      ia?x:ia_data;        adu[inp, out, ia, oa](receive(a,x), render(dc,x), ds) []
      inp?x:inp_data;      adu[inp, out, ia, oa](input(x,ds,a), echo(x,ds,a), ds)
endproc
```

5 Composition with the controller component

The controller applies temporal ordering constraints to the events offered at the gates of the ADU. Constraints are expressed concisely using the multi-way synchronisation of LOTOS, which allows the incremental composition of constraints. This specification style has been termed constraint oriented [18]. The controller process synchronises with the ADU on all the gates of the latter. The interactor, which is the parallel composition of the two, may only engage in events in the order that the controller component allows. Their composition is as follows:

```
adu [inp, out, ia, oa](initAbstraction, initDisplay, initDisplay)
        |[inp, out, ia, oa]|
controller [s, su, ab, inp, out, ia, oa]
```

The controller is started with a start event on gate *s*. This triggers process *run* which describes the run-time behaviour of the controller. Process *constraints* describes constraints on the order of the events offered on the gates of the ADU. Process *suspend* describes the suspension behaviour of the interactor: with the first *su* event interaction halts and with the second *su* event it resumes. The abort event *ab* terminates the operation of the interactor and the process exits. In the trivial case (below), *constraints* is a recursive process that allows all possible interaction sequences on the gates of the ADU.

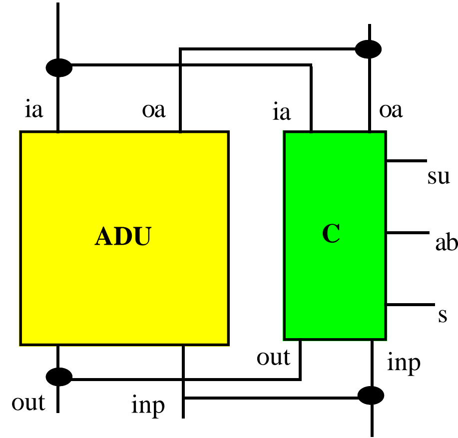

Fig. 4. Synchronisation of the ADU with the controller over all its gates

```
process controller [s, su, ab, inp, out, ia, oa] : exit :=
    s; run [su, ab, inp, out, ia, oa]
where

process run[su, ab, inp, out, ia, oa] : exit :=
    (constraints[inp, out, ia, oa]
            [>
    suspend [su, ab, inp, out, ia, oa])
where

process constraints[inp, out, ia, oa] : noexit :=
    ia?x:ia_data;           constraints[inp, out, ia, oa] []
    oa?x:oa_data;           constraints[inp, out, ia, oa] []
    out?x:disp;             constraints[inp, out, ia, oa] []
    inp?x:inp_data;         constraints[inp, out, ia, oa]
endproc

process suspend [su, ab, inp, out, ia, oa]: exit :=
    su;     (su; run[su, ab, inp, out, ia, oa]
            [] ab; exit)
    [] ab; exit
endproc
```

Alternative behaviours can be defined by modifying the *constraints* component of the controller. To put the constraint on the ADU that any data received from the application side is rendered instantly and that any input by the user is echoed instantly, the *constraints* process of the controller need only change as follows (changes are italicised):

```
process constraints[inp, out, ia, oa] : noexit :=
    ia?x:ia_data; out?y:disp;       constraints[inp, out, ia, oa] []
    oa?x:oa_data;                   constraints[inp, out, ia, oa] []
    out?x:disp;                     constraints[inp, out, ia, oa] []
    inp?x:inp_data;out?y:disp;      constraints[inp, out, ia, oa]
endproc
```

The interface designer may wish to distinguish the role of trigger events: output to the display or towards the application may be allowed only after a trigger event *out_t* ,*inp_t* respectively. For the ADU, the trigger event is simply treated as one more input gate from user or application. The process *constraints* is then written as follows:

```
process constraints[inp, out, ia, oa, inp_t, out_t] : noexit :=
      ia?x:ia_data;                          constraints[inp, out, ia, oa, inp_t, out_t] []
      out_t?x:ia_data; out?y:disp;           constraints[inp, out, ia, oa, inp_t, out_t] []
      inp?x:inp_data;                        constraints[inp, out, ia, oa, inp_t, out_t] []
      inp_t?x:inp_data; oa?y:oa_data;        constraints[inp, out, ia, oa, inp_t, out_t]
endproc
```

The advantages of this modular description of the controller are further illustrated, if, for example, one chooses to modify the role of triggering events, to demand that an input trigger does not affect the output behaviour and correspondingly the output trigger does not affect the input behaviour. Incidentally, this is the behaviour of the Pisa interactor which after an output trigger event may still receive data or an input trigger from the user.

```
process constraints[inp, out, ia, oa, inp_t, out_t] : noexit :=
      (ia?x:ia_data;                         constraints[inp, out, ia, oa, inp_t, out_t] []
      out_t?x:ia_data; out?y:disp;           constraints[inp, out, ia, oa, inp_t, out_t])|||
      (inp?x:inp_data; out?y:disp;           constraints[inp, out, ia, oa, inp_t, out_t] []
      inp_t?x:inp_data; oa?y:oa_data;        constraints[inp, out, ia, oa, inp_t, out_t])
endproc
```

This presentation has focused on the specification of single interactors. In fact, the modular controller component is expected to provide more benefits when forming compositions of interactors. The user interface can be modelled as a single interactor but also as a network of interactors. Compositions of interactors can be of two forms:

- the output from one interactor is directed to an input gate of the other. For such a composition to be possible, it is required that the intersection of the two domains of data offered over the connected gates is not empty. This type of connection works in almost the same way as has been demonstrated with the Pisa model [15,17].
- the behaviours of the interactors are composed into more complex ones e.g. their interleaving, their sequence etc. using the standard LOTOS process composition operators. Present work is investigating the conditions under which the ADC structure is preserved during these compositions.

6 A simple example

The example that follows demonstrates some of the ideas mentioned so far, namely, how interactor specifications are derived from the ADC model by instantiating the *ad* data type, how they are composed and, further, how the composition of two or more ADC interactors is an ADC interactor itself. The specification of a scrolling list which may contain any type of items e.g. icons, strings etc. is examined. The list is observed through a window whose contents depend on the window size and the position of the window relative to the displayed list, e.g. the index of the first displayed element. The user may scroll up and down the list by using a scrollbar.

The scrolling list is defined as the composition of a *scrollbar* interactor and a window display for the list, which will be referred to as a *list* interactor. Interactions are started,

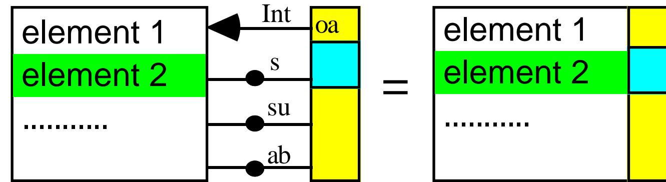

Fig. 5. A scrolling list as a composite interactor

suspended and terminated together, since they form part of the same interaction task, so the interactors synchronise on gates *s, su* and *ab.* The scrollbar receives input from the user as a cursor position, and converts this input to an integer value which is passed to the *list.* The integer value is passed via gate *oa* of the scrollbar interactor as an input to gate *inp* of *list* (figure 5). The latter interprets the integer value to produce a new starting position for the window thus achieving the effect of scrolling.
The data type *ls_ad* is the instantiation of the data type *ad* for the *list* interactor. It uses the data types *lstElements,* which models a list of elements, and *windowAndSelection,* which models the window display for the list. Their specifications include only those aspects of the operations that are necessary for the interactor specifications that follow.

```
type lstElements is Integer
sorts
    lstel, el
opns
    sel       :    lstel, Int    ->    lstel
    setstart  :    lstel, Int    ->    lstel
    which     :    lstel         ->    el
    wnstart   :    lstel         ->    Int
eqns
forall aList:lstel, N,M:Int
ofsort Int
    wnstart(setstart(aList, N)) = N;
    wnstart(sel(aList, N)) = wnstart(aList);
ofsort lstel
    sel(sel(aList,N),M) = sel(aList, M);
    sel(setstart(aList, N), M) = sel(aList,M);
ofsort el
    which(setstart(aList, M)) = which(aList);
endtype

type windowAndSelection is Integer, graphics
sorts
    window
opns
    mkWin        :    rectangle, Int       ->    window
    changeLne    :    window, Int          ->    window
    changeRect   :    window, rectangle    ->    window
    pick         :    window, pnt          ->    Int
    rect         :    window               ->    rectangle
    line         :    window               ->    Int
eqns
```

```
      forall index:Int, rct:rectangle, win:window, p:pnt
   ofsort rectangle
      rect(mkWin(rct,index)) = rct;
      rect(changeRect(win,rct)) = rct;
      rect(changeLne(win,index)) = rect(win);
   ofsort Int
      line(mkWin(rct,index)) = index;
      line(changeRect(win, rct)) = line(win);
      line(changeLne(win,index)) = index;
   endtype
```

The type *lstElements* defines an enquiry operator *which(aList)* that returns the selected element of a list. The selection of an element with index N for *lstElements* can be set by *sel(aList, N)*. Type *windowAndSelection* is associated with operation *pick(win, p)* which returns an index of the displayed window *win* (i.e. a line or icon number) given a mouse position *p*. Finally, the position of the window with respect to the list, can be set to N, by operation *setstart(aList, N)* which will use the integer value sent from the scroll bar.

```
   type ls_ad is lstElements, windowAndSelection
   opns
      inputPnt:        pnt, window, lstel        ->     lstel
      echoPnt:         pnt, window, lstel        ->     window
      inputNumber:     Int, window, lstel        ->     lstel
      echoNumber:      Int, window, lstel        ->     window
      render    :      window, lstel             ->     window
      receive :        lstel, lstel              ->     lstel
      scrListResult:   lstel                     ->     el
   eqns
   forall m: Int, p:pnt, w:window, aList,lstOld,lstNew:lstel
   ofsort  lstel
      inputPnt(p, w, aList) = sel(aList,pick(w,p));
      inputNumber(m, w, aList) = setstart(aList,m);
      receive(lstOld, lstNew) = lstNew;
   ofsort window
      echoPnt(p, w, aList) = changeLne(w, pick(w,p));
      render(render(w, lstOld), lstNew) = render(w,lstNew);
   ofsort el
      scrListResult(aList) = which(aList);
   endtype
```

Similarly, *scr_ad* defines the scrollbar functionality in terms of a graphical abstraction *scrBar* and a bounded value abstraction *boundedVal* (omitted for brevity). The enquiry operator *val(boundValue)* returns an integer result for the interactor.

```
   type scr_ad is  scrBar, boundedVal
   opns
      input     :      pnt, scrollbar, boundValue        ->     boundValue
      echo      :      pnt, scrollbar, boundValue        ->     scrollbar
      render    :      scrollbar, boundValue             ->     scrollbar
      render    :      scrollbar, rectangle              ->     scrollbar
      receive   :      boundValue, boundValue            ->     boundValue
      result    :      boundValue                        ->     Int
```

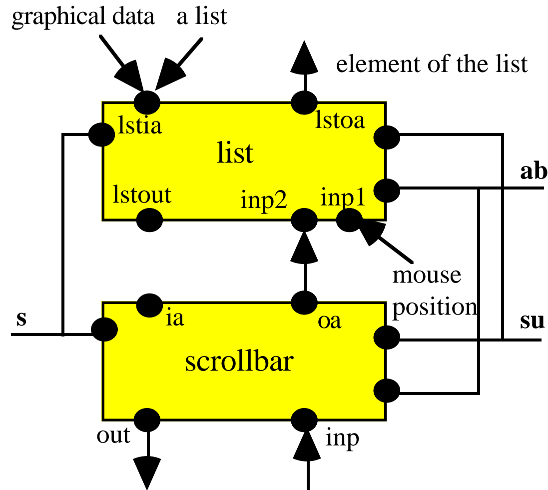

Fig. 6. The composition of the two interactors

```
eqns
forall  r:rectangle, p:pnt, sb: scrollbar,  bv1,bv2: boundValue
ofsort boundValue
      receive(bv1, bv2) = bv2;
ofsort scrollbar
      echo(p,sb,bv1) = changePnt(sb, p);
      render(sb, r) = changeRect(sb, r);
      render(sb, input(p,sb, bv1))=changePnt(sb,p);
ofsort Int
      result(bv1)=val(bv1);
endtype
```

The ADU for the *list* interactor is as follows

```
process adu[inp1, inp2, out, ia, oa](a:lstel, dc, ds: window) : noexit :=
      oa!scrListResult(a); adu[inp1, inp2, out, ia, oa](a, dc, ds) []
      out!dc; adu[inp1, inp2, out, ia, oa](a, dc, dc) []
      ia?x:lstel; adu[inp1, inp2, out, ia, oa](receive(a,x),render(dc,x), ds)[]
      inp1?x:pnt; adu[inp1, inp2, out, ia, oa](inputPnt(x,ds,a),echoPnt(x,ds,a), ds) []
      inp2?x:Int; adu[inp1,inp2,out,ia,oa](inputNumber(x,ds,a),echoNumber(x,ds,a), ds)
endproc
```

The controller is the same as in the general case; only process *constraints* needs to be modified with the sort identifiers of data type *ls_ad*. The ADU and the controller for the scrollbar interactor are again straightforward instantiations of the general model and are omitted for brevity. The composition of the two interactors (figure 6), is written as follows (note the renaming to *oa* of gate *inp* in the instantiation of the interactor *lst*):

```
scr[s, su, ab, inp, out, a, oa] (initBV, initSB, initSB)
      |[s, su, ab, oa]|
list[s, su, ab, oa, lstout, lstia, lstoa] (initLst, initWindow, initWindow)
```

By its construction the ADC model is compositional: the result of the composition of two ADC interactors is an ADC interactor. For example, the composition of the previous section is equivalent to an ADC which is of the form:

```
adu[inp, out, ia, oa, lstout, lstia, lstoa]
        |[inp, out, ia, oa, lstout, lstia, lstoa]|
controller [s, su, ab, inp, out, ia, oa, lstout, lstia, lstoa]
```

where *adu* is defined as

```
scr_adu [inp, out, ia, oa](initBV, initSB, initSB)
        |[oa]|
lst_adu[oa, lstout, lstia, lstoa](initLst, initWindow, initWindow)
```

and *controller* is defined as

```
scr_controller[s, su, ab, inp, out, ia, oa]
        |[s, su, ab, oa]|
lst_controller[s, su, ab, oa, lstout, lstia, lstoa]
```

7 Expression of interface properties within the ADC model

Analytic expressions of properties of interaction pertaining to the usability of interactive systems are presented below. Novel characterisations of interaction are not proposed; the presentation concentrates on expressions for properties of interaction introduced previously within other formal models. The intention is to suggest the analytic potential of the ADC model. Properties of interactive systems are examined below in two categories:

- logical properties of interactor behaviour. This category covers safety and liveness properties of the interface specification similar to those discussed in [14, 16, 18].
- properties of the correspondence between the information displayed and that sent to the application. These properties correspond to what has been termed 'result display properties' which have been discussed within various formal models e.g. the Interactive Processes model [19] the Agents model [1] and the red-PIE model [5]

First, a few concepts regarding LOTOS behaviour expressions and a generalised description of the gate sets of the ADC interactor are introduced.

7.1 A formal model for the study of interactors

Given a process B in LOTOS, a set of labelled transitions may be derived. These are denoted as $B - \alpha \rightarrow C$, where α is an action (label), and C is another behaviour expression. A LOTOS action declaration has the form g?x:t, where x is a variable and t is a sort identifier which indicates the domain of values over which x ranges. For example, g?x:integer specifies a set of actions $g<v>$ where $<v>$ is in the domain of the integers.

A sequence of unobservable internal actions, represented below by ε, may effect a transition from B to C denoted as $B = \varepsilon \Rightarrow C$. A second transition relation, needed to discuss the observed behaviour of transition systems, is defined as:

$$B = a \Rightarrow C \quad \textit{iff} \quad B = \varepsilon \Rightarrow B_1 - a \rightarrow B_2 = \varepsilon \Rightarrow C$$

A trace is a sequence of observable actions in which the process may successfully participate starting from its initial state. The set of traces of a process B, denoted $Tr(B)$,

is by its definition prefix closed, i.e. every prefix of a trace also belongs to the set of traces. A trace $\sigma \in \mathrm{Tr}(B)$, defines a sequence of transitions as follows:

$$B = \sigma \Rightarrow C \ iff \ \sigma = a_1..a_n \ and \ B = a_1 \Rightarrow B_1 = a_2 \Rightarrow \ldots = a_n \Rightarrow C$$

For a given behaviour expression B, the set of outgoing transitions, i.e. all the actions for which a transition is possible will be denoted as $out(B)$.

A trace that cannot be extended because it is infinite or it has no outgoing transitions is called a *full trace*. Let $\mu Tr(B)$ denote the set of *full traces* of the process B. If a trace σ is finite then the last element of the trace is $last(\sigma)$, and $b \ in \ \sigma$ will denote that an action b is in the trace.

Topology of interaction gates. The interaction model was introduced with only one gate for input from lower levels, one for display etc. For the more general discussion that follows it is more convenient to talk of sets of such gates as there could be more than one gates of each category.

The set of gates of the interactor can be partitioned into the set of input/output gates G_{io} and the control gates G_c.

$$G = G_c \cup G_{io} \text{ and } G_c \cap G_{io} = \emptyset, \ G_{io} \neq \emptyset$$

G_c consists of the subsets $G_{start}, G_{suspend}$ and G_{abort} where start, suspend and abort events respectively will be observed.

$$G_c = G_{start} \cup G_{suspend} \cup G_{abort}$$

G_{io} is the non-empty set of input and output gates of the interactor. It is further partitioned to two non-empty sets G_a and G_u, that communicate with higher and lower levels of abstraction respectively.

$$G_{io} = G_a \cup G_u \text{ and } G_a \cap G_u = \emptyset, \ G_a, G_u \neq \emptyset$$
$$G_a = G_{oa} \cup G_{ia} \text{ and } G_{oa} \cap G_{ia} = \emptyset$$
$$G_u = G_{inp} \cup G_{out} \text{ and } G_{inp} \cap G_{out} = \emptyset$$

7.2 Logical Properties

Input and output correctness, restartability, undo and properties of the connection of the interactors are examined under this heading. Some representative examples below demonstrate how such requirements can be expressed in terms of the ADC model. In the following UI denotes the initial state of the user interface described as an ADC interactor.

- Every input is echoed immediately

  $If \ UI = \sigma \Rightarrow UI' \wedge last(\sigma) = a?x{:}inp_data \wedge a \in G_{inp} \ then$

  $\exists b \in G_{out}, D{:}disp, A{:}abs \mid out(hide \ G_{abort} \cup G_{suspend} \ in \ UI') = \{b!echo(x, D, A)\}$
- Every user input is echoed eventually

  $If \ UI = \sigma \Rightarrow UI' \wedge last(\sigma) = a?x{:}inp_data \wedge a \in G_{inp} \ then$

  $\forall \vartheta \in \mu Tr(UI') \bullet \exists b \in G_{out}, D{:}disp, A{:}abs \mid b!echo(x, D, A) \ in \ \vartheta$
- A command sequence is *restartable* if it is possible to extend it so that it returns to the initial state.

  $\forall \mu \in Tr(UI) \mid UI = \mu \Rightarrow UI' \bullet \exists \nu \in Tr(UI') \mid UI' = \nu \Rightarrow UI$
- Any command c followed by *undo* should leave the system in the same state as before the command (single step undo).

  $\forall \sigma \in Tr(UI) \mid UI = \sigma \Rightarrow UI' \bullet if \ UI' = c \Rightarrow UI'' \ then \ UI'' = undo \Rightarrow UI'$

Sufrin and He [19] noted that this definition for undo is unrealistically strict so they proposed the concepts of *weak* and *strong undo*. Their expression requires the

Display Predictability	$out(P_1) = out(P_2) \Rightarrow P_1 = P_2$
Result Predictability	$out(R_1) = out(R_2) \Rightarrow R_1 = R_2$
Honesty	$out(P_1) = out(P_2) \Rightarrow out(R_1) = out(R_2)$
Trustworthiness	$P_1 = P_2 \Rightarrow out(R_1) = out(R_2)$
WYSIWYG(weak)	$out(P_1) = out(P_2) \Rightarrow R_1 = R_2$
WYSIWYG(strong)	$P_1 = P_2 \Rightarrow R_1 = R_2$
Goal defines view	$out(R_1) = out(R_2) \Rightarrow P_1 = P_2$

Fig. 7. Expressions of some usability related properties

characterisation of application and user view of an interactor behaviour which is defined in the next section.

7.3 Result-Display Relationships

Sufrin and He [19] use the concepts of equivalence and indistinguishability of *results* and *views* to classify a set of usability related properties of interactive systems. In that context, *result* referred to the part of the application state that is relevant to the users' goals, and *view* to the part of the state that is made perceivable to the user. Such entities would be called equivalent if they were the same after a particular input sequence and indistinguishable if no further experimentation by the user, could betray any difference between them. Abowd [1] used the same classification adapting the definitions to his Agent model. He compared *results* and *displays* which were defined as restrictions on the internal state of an Agent. In general, they reflect the intuition that during interaction the user is only made aware of what is displayed and uses that to 'plan' her subsequent interactions. The user inputs have distinct effects on what is made perceivable and what is the effect of the interaction on the internal state of the system.

The same intuition, applied to this more architecture oriented model, requires the identification of two views of the interactor. One view of the interactor is from the system side and the other from the user side. The former consists in the behaviour of the interactor restricted to the output gates toward the system G_{oa} (the data that the system sends to the interactor is not of interest in this context, unless as a source of non-determinism). The user's view of the interactor can be defined as the interactor behaviour restricted over gates G_u, i.e. both input and output gates on the user side.

Consider the processes P and R defined with the pseudo-LOTOS expressions below.

$$P \;=\; hide\ G_c \cup G_a\ in\ UI \qquad\qquad R \;=\; hide\ (G - G_{oa})\ in\ UI$$

The corresponding notions to those of equivalence and indistinguishability mentioned above, are quite intuitive within the process algebraic framework adopted. The observable behaviour of processes are compared with respect to two aspects: the events they offer to participate in, denoted by $out(Q)$ for a process Q and their behaviour as might be observed with subsequent experimentation. The latter concept has been formalised as the testing equivalence between processes [4] denoted by the equals sign (e.g. $P=Q$).

A summary classification of these properties similar to [1, pp.160] can now be written as in figure 7. Processes P and R are also useful in expressing refinements of restartability, undo properties and for expressing the predictability of a single command, as in [19]. Such expressions like those of figure 7, are useful intuitive aids for the designer, but in practice may prove impossible to verify automatically. For

specifications like the scrolling list example, it is easy to see how the sets *out(P)* and *out(R)* may be infinitely large or that processes P and R may not be finite.

8 Conclusion - Future Work

The ADC interactor model introduced above is a development of the interactor model of Paterno' and Faconti [15, 17]. Its primary use is intended to be constructive for the formal specification of user interface designs. It may also be used analytically as is suggested by the exposition of a wide range of properties of interaction in section 7.
A formal description of the model was given in LOTOS. The use of a constraint based style of specification enabled the separation of the dialogue and the data transforming aspects of the interactor. This modularity was introduced to make the model more usable as a design notation. The ADC interactor can be used in a similar way to the Pisa model to describe particular interaction styles and to construct interface specifications as a composition of elementary interactors. Dialogue specification is localised within the control component; in fact a controller may be associated with a composition of interactors imposing constraints on their global externally observable behaviour. One of the aims of this research is to support the use of user task knowledge in the design of interactive systems. Current work is examining how such control information can be related to task knowledge. Previous research [11] has focused on representations of device independent task knowledge of users without reference to a particular system model and has attempted to prescribe relationships between task and system models [12]. The aim of this research is to provide a theoretical understanding of task based design and a practical way of embedding this into a design method.
The definition of properties of interaction in terms of the ADC interactor model is motivated by the goal of supporting automatic verification of interface designs. From a practical point of view it is interesting to examine the feasibility of supporting their automatic verification. To this end, a few research tools for the automated verification of LOTOS specifications are being experimented with. Two classes of verification techniques are of practical interest. One is the use of logical specifications of such properties as demonstrated in [17, 18]. The second approach is to construct behavioural specifications where the properties are described directly as higher level LOTOS specifications. Given appropriate abstraction criteria, the verification of a property amounts to verifying equivalence of an abstraction of the design specification with the property specification, using a verification tool such as [8]. This research is still in its early days; it is hoped that automatic verification of user interface properties will make research results from the application of formal methods to human computer interaction more usable within a design context.

Acknowledgements

Many thanks to Stephanie Wilson and to Jon Rowson for their comments and for proof reading this paper.

References

1. Abowd G.D.: Formal Aspects of Human Computer Interaction, PhD thesis, University of Oxford, Technical Report YCS 161, University of York (1992).

2. Bolognesi T., Brinksma E.: Introduction to the ISO specification language LOTOS. In: Van Eijk P., Vissers C., Diaz M. (eds.): The Formal Description Technique LOTOS, Elsevier Science Publishers BV (1989), 23-73.

3. Coutaz J.: PAC, an Object Oriented Model for Dialog Design. In: Bullinger H.J., Shakiel B. (eds.): Human Computer Interaction - INTERACT-'87, Elsevier Science Publishers BV (1987), 431-436.

4. De Nicola R. , Hennessy M.C.B.: Testing Equivalence for Processes. Theoretical Computer Science, North Holland, Vol. 34, 83-133 (1984).

5. Dix A.J.: Formal Methods for Interactive Systems, Academic Press (1991).

6. Duke D.J., Harisson M.D.: Abstract Interaction Objects. In: Hubbold R.J., Juan R. (eds.): Eurographics'93, Computer Graphics Forum, Vol. 12, No. 3, 26-36 (1993).

7. Faconti G.P.: Towards the Concept of Interactor. Amodeus Project Document: System Modelling/WP8 (1993).

8. Fernadez J.C., Garavel H., Mounier L., Rasse A., Rodriguez C., Sifakis J.: A toolbox for the verification of LOTOS Programs. In: 14th International Conference on Software Engineering, Melbourne, May (1992).

9. Harrison M.D., Dix A.J.: A state model of direct manipulation in interactive systems. In: Harisson M. D., Thimbleby H.W. (eds.): Formal Methods in Human Computer Interaction, Cambridge Univ. Press (1990), 129-151.

10. Krasner G.E., Pope S.T.: A Cookbook For Using the Model-View-Controller User Interface Paradigm in The Smalltalk-80 System, Journal of Object Oriented Programming, Vol.1, No.3, 26-49 (1988).

11. Markopoulos P., Wilson S., Johnson P.: Representation and Use of Task Knowledge in a User Interface Design Environment. IEE Proceedings~E, Computers and Digital Techniques, Vol.141, No.2, 79-84 (1994).

12. Markopoulos P., Gikas S.: Towards A Formal Model For Extant Task Knowledge Representation. In: Stary C (ed.): 1st Interdisciplinary Workshop on Cognitive Modelling and User Interface Development, Vienna (1994).

13. Myers B.A.: A New Model for Handling Input. ACM Transactions on Information Systems, Vol.8, No.3, 289-320 (1990).

14. Palanque P., Bastide R.: Petri net based design of user driven interfaces using the interactive cooperative objects formalism. In: Paterno' F. (ed.): Design Specification and Verification of Interactive Systems, Eurographics workshop, 215-228 (1994).

15. Paterno' F., Faconti G.: On the use of LOTOS to describe graphical interaction. In: Monk A., Diaper D., Harrison M.D., (eds.): People and Computers VII, Proc. HCI'92 Conference, Cambridge Univ. Press (1992), 155-173.

16. Paterno' F.: Definition of properties of user interfaces using action based temporal logic. In: Proceedings, 5th conference in Software Engineering and Knowledge Engineering (1993), 314-318.

17. Paterno' F.: A Theory of User Interaction Objects. Journal of Visual Languages and Computing, Academic Press Ltd, Vol. 5, 227-249 (1994).

18 Paterno' F., Mezzanotte M.: Analysing Matis by Interactors and ACTL. Amodeus Project Document: System Modelling/WP36 (1994).

19. Sufrin B., He J.: Specification analysis and refinement of interactive processes. In: Harisson M.D., Thimbleby H.W. (eds.): Formal Methods in Human Computer Interaction, Cambridge Univ. Press (1990), 153-200.

20. Vissers C.A., Scollo G., van Sinderen M., Brinksma E.: Specification styles in distributed systems design and verification. Theoretical Computer Science Vol. 89, 179-206 (1991).